# ARCHITECTURE FOR AN INDEXER

## BACKGROUND OF THE INVENTION

### 1. Field of the Invention

**[0001]** The present invention is related generally to an architecture for an indexer and more particularly to an architecture for an indexer that indexes tokens that may be variable-length, where variable-length data may be attached to at least one token occurrence.

### 2. Description of the Related Art

**[0002]** The World Wide Web (also known as WWW or the "Web") is a collection of some Internet servers that support Web pages that may include links to other Web pages. A Uniform Resource Locator (URL) indicates a location of a Web page. Also, each Web page may contain, for example, text, graphics, audio, and/or video content. For example, a first Web page may contain a link to a second Web page.

**[0003]** A Web browser is a software application that is used to locate and display Web pages. Currently, there are billions of Web pages on the Web.

**[0004]** Web search engines are used to retrieve Web pages on the Web based on some criteria (e.g., entered via the Web browser). That is, Web search engines are designed to return relevant Web pages given a keyword query. For example, the query "HR" issued against a company intranet search engine is expected to return relevant pages in the intranet that are related to Human Resources (HR). The Web search engine uses indexing techniques that relate search terms (e.g., keywords) to Web pages.

**[0005]** An important problem today is searching large data sets, such as the Web, large collections of text, genomic information, and databases. The underlying operation that is needed for searching is the creation of large indices of tokens quickly and efficiently. These indices, also called inverted files, contain a mapping of tokens, which may be terms in text or more abstract objects, to their locations, where a location may be a document, a page number, or some other more abstract notion of location. The indexing

problems is well-known and nearly all solutions are based on sorting all tokens in the data set. However, many conventional sorting techniques are inefficient.

**[0006]** Thus, there is a need for improved indexing techniques.

## SUMMARY OF THE INVENTION

**[0007]** Provided are a method, system, and program for indexing data. A token is received. It is determined whether a data field associated with the token is a fixed width. When the data field is a fixed width, the token is designated as one for which fixed width sort is to be performed. When the data field is a variable length, the token is designated as one for which a variable width sort is to be performed.

## BRIEF DESCRIPTION OF THE DRAWINGS

Referring now to the drawings in which like reference numbers represent corresponding parts throughout:

FIG. 1 illustrates, in a block diagram, a computing environment in accordance with certain implementations of the invention.

FIG. 2 illustrates logic implemented prior to generation of an index in accordance with certain implementations of the invention.

FIGs. 3A, 3B, and 3C illustrate logic implemented for an index build phase in accordance with certain implementations of the invention.

FIG. 3D illustrates multi-threading for performing index build runs in accordance with certain implementations of the invention.

FIG. 3E illustrates a structure of a sort key in accordance with certain implementations of the invention.

FIG. 4A illustrates logic implemented for an index merge phase in accordance with certain implementations of the invention.

FIG. 4B illustrates threading for merging sorted runs in accordance with certain implementations of the invention.

FIG. 4C illustrates output of an indexing component in accordance with certain implementations of the invention.

FIG. 5A illustrates a table of benchmark test results for index build threading and hyperthreading in accordance with certain implementations of the invention.

FIG. 5B illustrates a table of index build scaling performances in accordance with certain implementations of the invention.

FIG. 6 illustrates an architecture of a computer system that may be used in accordance with certain implementations of the invention.

## DETAILED DESCRIPTION

**[0008]** In the following description, reference is made to the accompanying drawings which form a part hereof and which illustrate several implementations of the present invention. It is understood that other implementations may be utilized and structural and operational changes may be made without departing from the scope of the present invention.

**[0009]** Implementations of the invention provide a fast technique for creating indices by using a dual-path approach in which the task of sorting (indexing) is broken into two separate sort processes for fixed width and variable width data.

**[0010]** FIG. 1 illustrates, in a block diagram, a computing environment in accordance with certain implementations of the invention. A client computer 100 is connected via a network 190 to a server computer 120. The client computer 100 may comprise any computing device known in the art, such as a server, mainframe, workstation, personal computer, hand held computer, laptop telephony device, network appliance, etc. The network 190 may comprise any type of network, such as, for example, a Storage Area Network (SAN), a Local Area Network (LAN), Wide Area Network (WAN), the Internet, an Intranet, etc. The client computer 100 includes system memory 104, which may be implemented in volatile and/or non-volatile devices. One or more client applications 110 and a viewer application 112 may execute in the system memory 104. The viewer application 112 provides an interface that enables searching of a set of documents (e.g., stored in one or more data stores 170. In certain implementations, the viewer application 112 is a Web browser.

**[0011]** The server computer 120 includes system memory 122, which may be implemented in volatile and/or non-volatile devices. A search engine 130 executes in the system memory 122. In certain implementations, the search engine includes a crawler component 132, a static rank component 134, a duplicate detection component 138, an anchor text component 140, an indexing component 142, and a tokenizer 144. Although components 132, 134, 138, 140, 142, and 144 are illustrated as separate components, the functionality of components 132, 134, 138, 140, 142, and 144 may be implemented in fewer or more or different components than illustrated. Additionally, the functionality of the components 132, 134, 138, 140, 142, and 144 may be implemented at a Web application server computer or other server computer that is connected to the server computer 120. Additionally, one or more server applications 160 execute in system memory 122. System memory 122 also includes one or more in-memory sort buffers 150.

**[0012]** The server computer 120 provides the client computer 100 with access to data in at least one data store 170 (e.g., a database). Although a single data store 170 is illustrated, for ease of understanding, data in data store 170 may be stored in data stores at other computers connected to server computer 120.

**[0013]** Also, an operator console 180 executes one or more applications 182 and is used to access the server computer 120 and the data store 170.

**[0014]** The data store 170 may comprise an array of storage devices, such as Direct Access Storage Devices (DASDs), Just a Bunch of Disks (JBOD), Redundant Array of Independent Disks (RAID), virtualization device, etc. The data store 170 includes data that is used with certain implementations of the invention.

**[0015]** The goal of text indexing is to take an input document collection (a "document-centric" view) and produce a term-centric view of the same data. In the term-centric view, data is organized by term, and for each term, there is a list of occurrences (also referred to as postings). A posting for a term consists of a list of the documents and offsets within the document that contain the term. In addition, it is useful (e.g., for ranking) to attach some extra attributes to each term occurrence. Some example

attributes include, whether the term occurs in a title, occurs in anchor text or is capitalized.

**[0016]** Indexing is equivalent to a sorting problem, where a primary sort key is a term and a secondary sort key is an offset in the document. For example, document D1 contains "This is a test", document D2 contains "Is this a test", and document D3 contains "This is not a test". Table A illustrates term information in tabular form, ordered as the terms would be read in from documents D1, D2, and D3 (in that order), in accordance with certain implementations of the invention.

Table A

| Term | Document Identifier | Offset | Data |
|---|---|---|---|
| this | 1 | 0 | 1 |
| is | 1 | 1 | 0 |
| a | 1 | 2 | 0 |
| test | 1 | 3 | 0 |
| is | 2 | 0 | 1 |
| this | 2 | 1 | 0 |
| a | 2 | 2 | 0 |
| test | 2 | 3 | 0 |
| this | 3 | 0 | 1 |
| is | 3 | 1 | 0 |
| not | 3 | 2 | 0 |
| a | 3 | 3 | 0 |
| test | 3 | 4 | 0 |

**[0017]** The first column in Table A is the term, the second column is the document identifier, and the third column is the offset in the document. fourth column is a data field, and, in this example, the data field indicates whether a term is case-folded (capitalized), where 1 is used to indicate that the term is case-folded and a 0 indicates that the term is not case-folded.

**[0018]** Sorting is also accompanied by a compression, where the final index is written in a compact, easily searchable form. Table B illustrates sorted terms from Table A in accordance with certain implementations of the invention.

Table B

| Term | Document | Offset | Data |
|---|---|---|---|
| a | 1 | 2 | 0 |
| a | 2 | 2 | 0 |
| a | 3 | 3 | 0 |
| is | 1 | 1 | 0 |
| is | 2 | 0 | 1 |
| is | 3 | 1 | 0 |
| not | 3 | 2 | 0 |
| test | 1 | 3 | 0 |
| test | 2 | 3 | 0 |
| test | 3 | 4 | 0 |
| this | 1 | 0 | 1 |
| this | 2 | 1 | 0 |
| this | 3 | 0 | 1 |

**[0019]** Posting lists may be represented more compactly by grouping together occurrences of a term in a list as illustrated in Table C:

Table C

| | |
|---|---|
| a | (1,2,0), (2, 2,0), (3,3,0) |
| is | (1,1,0), (2,0,1), (3,1,0) |
| not | (3,2,0) |
| test | (1,3,0), (2,3,0), (3, 4, 0) |
| this | (1,0,1), (2,1,0), (3,0,1) |

For example, one posting list is: this (1,0,1), (2,1,0), (3,0,1). The posting lists may be stored in a compressed binary format, but, for illustration, the logical view of each posting list is provided. In certain implementations, the document identifier and offset values in the posting lists are delta encoded and then compressed.

**[0020]** FIG. 2 illustrates logic implemented before generating an index in accordance with certain implementations of the invention. Control begins at block 200 and documents that are to be indexed by the search engine 130 are obtained. In certain implementations, the documents are published or pushed (e.g., as may be the case with newspaper articles) to the indexing component 142. In certain implementations, the crawler component 132 discovers, fetches, and stores the documents. In certain implementations, the crawler component 132 may discover documents based on, for example, certain criteria (e.g., documents were accessed within the last month). Additionally, the crawler component 132 may discover documents in one or more data stores connected directly (e.g., data store 170) or indirectly (e.g., connected to server computer 120 via another computing device (not shown)) to server computer 120. In certain implementations, the crawler component 132 discovers, fetches, and stores Web pages in data store 170.

**[0021]** The documents are bundled in files in data store 170. That is, when documents are retrieved by the crawler component 132, one or more documents are stored into a file,

called a bundle. The bundle includes as many documents as fit into the bundle, based on the size allocated for a bundle. In certain implementations, the bundle size defaults to 8MB (megabytes). Having bundles improves performance by allowing large batches of documents to be fetched at once using sequential I/O. Otherwise, if each of the documents was retrieved as a separate file, the seeks times would impact performance.

**[0022]** Doing a scan of the document store 170 is I/O bound. Each document in the data store contains a tokenized document content (e.g., stored in data vectors), and includes both an original token and a stemmed token. A tokenized document content may be described as breaking a document into individual terms with tokens determining word boundaries. An original token refers to an original term in a document (e.g., "running") that includes case-folding, while a stemmed token refers to a root for the original token (e.g., "run").

**[0023]** An arbitrary binary data field may be attached with each token occurrence. In certain implementations, this data field is a one byte attribute field for document text tokens. In certain implementations, the data field may be of varying width (e.g., for document metadata tokens). For instance, some metadata tokens include the document keyword count, document unique keyword count, document path (e.g., URL or hash), site path (e.g., URL or hash), and document rank.

**[0024]** The tokenized content is stored in the data store 170 for performance reasons. That is, tokenization and parsing are slow and CPU intensive. Once the bulk of documents have been crawled, there are typically small incremental updates to the data store 170. Thus, rather than re-running the tokenizer each time an index is to be built (e.g., once a day), the stored tokenized document content is used to build the index.

**[0025]** In block 202, the static rank component 134 reviews the stored documents and assigns a rank to the documents. The rank may be described as the importance of the source document relative to other documents that have been stored by the crawler component 132. Any type of ranking technique may be used. For example, documents that are accessed more frequently may receive a higher rank.

**[0026]** Thus, document ranks are computed prior to indexing, and the rank value is available at indexing time as part of a document. In certain implementations, the rank is an ordinal document rank (e.g., 1, 2, 3, 4, etc.), so as the documents come in, the document rank may be used as the document identifier. The ordinal document ranks are monotonically increasing, though they are not necessarily sequential (e.g., there may be gaps as documents are being scanned). Documents need not be fed to the indexing component 142 in rank order, instead, documents may be fed in using an arbitrary ordering.

**[0027]** In block 204, an anchor text component 140 extracts anchor text and builds a virtual document with the anchor text. In block 206, optionally, the duplicate detection component 138 may perform duplication detection to remove duplicate documents from data store 170. The duplicate detection processing may also occur as documents are being stored.

**[0028]** There are many variations on indexing and on compressing posting lists. One approach is a sort-merge approach, in which sets of data are read into memory, each set sorted, and each set copied to storage (e.g., disk), producing a series of sorted runs. The index is generated by merging the sorted runs.

**[0029]** In certain implementations, the indexing component 142 implements a sort–merge approach. The sort-merge approach provides good performance and is suitable for batch style indexing in which data is indexed at once. Internally, the indexing component 142 has two phases: index build runs and index merge.

**[0030]** FIGs. 3A, 3B, and 3C illustrate logic implemented for an index build phase in accordance with certain implementations of the invention. FIG. 3D illustrates multi-threading for performing index build runs in accordance with certain implementations of the invention. In FIG. 3A, control begins at block 300 with the indexing component 142 scanning documents in data store 170 to fill a first in-memory sort buffer. When data store 170 is capable of storing documents, the data store 170 may also be referred to as a "document store". In block 302, the indexing component 142 generates a sort key for each token in each document in the first in-memory sort buffer, starting with a first token

and forwards the sort key to either a second or a third in-memory sort buffer. FIG. 3E illustrates a structure of a sort key in accordance with certain implementations of the invention. The sort key may be a 128-bit sort key that is composed of a token identifier 360 in the upper 64 bits (8 bytes) and a location 366 in the lower 64 bits. Additionally, a fixed-width or variable width data field may be carried along with the sort key. The token identifier 360 includes a bit 362 that specifies a token type (e.g., content or metadata) and 63 bits that identify a token hash 364.

**[0031]** Although any hash function may be used with various implementations of the invention, one that is fast and that works well on large amounts of text tokens that have many common prefixes and suffixes is useful. In certain implementations, Pearson's hash function is used, which is described further in "Fast Hashing of Variable-Length Text Strings" by Peter K. Pearson, Communications of the ACM, 33(6):677-680, June 1990. Pearson's hash function is scalable.

**[0032]** The location 366 includes a 32 bit unique document identifier 368, a bit 370 that indicates whether the section associated with the sort key is content or anchor text, and a 31 bit offset 372 that identifies the offset of the term within the document. When a document is read from the data store 170, the location of the document is generated using the document rank and the token offset within the document. Since a full sort is later done on the location, documents may arrive in any order, and the indexing component 142 is able to reorder them properly in rank order. Also, multiple terms may occur at a same location (e.g., which is useful for adding metadata to mark terms or sequences of terms).

**[0033]** Thus, rather than having separate document identifier (ID) fields and offset fields, certain implementations of the invention use a global 64-bit location 366 to identify the locations of terms. With the use of this 64-bit location, up to 4 billion documents are supported in one index, with 4 billion locations in each document. In alternative implementations, other bit sizes may be used for the global location. Also, each posting occurrence may have arbitrary binary data attached to it (e.g., for ranking).

**[0034]** Looking at the bit encoding of the sort key, by sorting on the sort key, implementations of the invention simultaneously order the index by the token identifier. Additionally, the encoding of the sort key allows the content tokens to come before the metadata tokens (allowing the two paths for the content (fixed width) and metadata (variable width) tokens to be sorted independently and brought together in the merge phase). The encoding of the sort key, for each unique token identifier, allows occurrences to be ordered by document identifier, and, if the document identifier is also the document rank, then, the tokens are in document rank order. Anchor text and content portions of a document may be processed separately and brought together because both portions have the same document identifier. For document occurrences of each token, the offsets of the occurrences within the document are in occurrence order due to the sorting technique used (e.g., due to the stable sort property of a radix sort). Tokens within a document may be fed in order, while anchor documents may be fed in after the content documents.

**[0035]** In block 304, the indexing component 142 determines whether a sort key is for a fixed width sort. If so, processing continues to block 306, otherwise, precessing continues to block 316.

**[0036]** In block 306, the indexing component 142 forwards the sort key to a fixed width sort via the second in-memory sort buffer and processing loops back to block 302. In block 316, the indexing component 142 forwards the sort key to a variable width sort via the third in-memory sort buffer and processing loops back to block 302. If all documents in the first in-memory sort buffer have been processed, the indexing component 142 scans a new set of documents into the in-memory sort buffer for processing, until all documents have been processed.

**[0037]** In FIG. 3B, in block 308, the indexing component 142 determines whether the second in-memory sort buffer is full. If so, processing continues to block 310, otherwise, processing loops back to block 308. In certain implementations, in block 308, the indexing component 142 treats a sort buffer as full if all tokens in all documents have been processed, and so additional sort keys will not be added to the sort buffer. In block

310, the indexing component 142 performs a fixed width sort on the sort keys in the second in-memory sort buffer. In block 312, the indexing component 142 writes the sorted run to temporary storage. In block 314, the indexing component 142 resets the second in-memory sort buffer, which allows additional sort keys to be stored in the second in-memory sort buffer.

**[0038]** In FIG. 3C, in block 318, the indexing component 142 determines whether the third in-memory sort buffer is full. If so, processing continues to block 320, otherwise, processing loops back to block 318. In certain implementations, in block 308, the indexing component 142 treats a sort buffer as full if all tokens in all documents have been processed, and so additional sort keys will not be added to the sort buffer. In block 320, the indexing component 142 performs a variable width sort on the sort keys in the third in-memory sort buffer. In block 322, the indexing component 142 writes the sorted run to temporary storage. In block 324, the indexing component 142 resets the third in-memory sort buffer, which allows additional sort keys to be stored in the third in-memory sort buffer.

**[0039]** The in-memory sort buffer controls the size of the sorted runs, which in turn is related to the total number of sorted runs generated. In certain implementations, each in-memory sort buffer is as large as possible as some systems have low limits on the number of file descriptors that may be opened at once. In particular, in certain implementations, each sort run is written to disk as a separate file, and, at merge time, all of these files are opened in order to perform the merging. In certain operating systems, each file that is currently open uses up a file descriptor, and, in some cases, there is a limit to the number of file descriptors (and thus files) that can be open at one time by a process. Additionally, the indexing component 142 uses a linear time in-memory sort technique so that performance does not change as the in-memory sort buffer size increases. In certain implementations, the in-memory sort buffer size defaults to 1.5GB (gigabytes).

**[0040]** With reference to FIG. 3D, in a pipelined approach to indexing, pipelining speeds up indexing by allowing I/O and computation work to overlap. There are several places where the sort may be pipelined in implementations of the invention. First, prefetching

read threads 332, 334 may be used to read in data from the data store 330 and store the data in a queue 336. An indexer thread 338 scans tokens in the documents, hashes the tokens, generates sort keys, and accumulates sort keys in a sort buffer. The indexer thread 338 forwards a sort key that is for content (i.e., fixed width data) to an indexer sort and forwards a sort key that is for metadata (i.e., variable width data) to a disk sort 342. The indexer sort 340 performs some processing and stores the sort keys in queue 344, and when queue 344 is full, a radix sort thread 348 performs a fixed width sort of the sort keys and stores the sorted keys as a sorted run in a temporary data store 352. The disk sort 346 performs some processing and stores the sort keys in queue 346, and when queue 346 is full, a radix sort thread 350 performs a variable width sort of the sort keys and stores the sorted keys as a sorted run in a temporary data store 354.

**[0041]** The indexing component 142 takes advantage of the fact that when inserting tokens of a document into an in-memory sort buffer, the locations of the tokens are being inserted in order. Since radix sort is a stable sort that preserves the inserted order of elements when there are ties, a 96-bit radix sort may be used, since the tokens within a document are inserted in order (these are the lower 32 bits of the sort key). Additionally, if there were no document location remapping, a 64-bit radix sort may be used. Because documents are inserted in arbitrary rank, and thus arbitrary document identifier order, a 96-bit radix sort is used.

**[0042]** Thus, when a sort buffer 344, 346 is full, the sort buffer 344, 346 is handed off to an appropriate sort thread that performs radix sort and writes the sorted run to storage.

**[0043]** In order to use the pipelined, parallel processing technique of FIG. 3D, the indexing component 142 replaces variable width tokens with a fixed width token identifier (e.g., by assigning a unique sequentially generated identifier number or using a hashing function). That is, the variable width tokens are transformed into fixed width tokens. When a uniquely generated identifier 360 is used, a mapping of terms to identifiers is maintained so that as terms are indexed, the terms are replaced with their identifiers. A hash function using a large hash space is unlikely to have collisions. In certain implementations, a 63-bit hash (reserving the upper for marking special

meta-tokens) may be used. In addition, with use of the 63-bit hash, certain testing indicated that there were no collisions among 260 million terms. In certain alternative implementations, a hash function approach using a minimal perfect hash function that ensures that there are no collisions may be used.

**[0044]** In practice, a large percent (e.g., 99.9%) of the data fields are one byte long, while the remaining data fields are of variable width. Thus, the sorting may be separated into two sorts: one where the data fields are one byte and one where the data fields are variable width. This results in an optimizable fixed width field sorting problem in which a 128-bit sort key is composed of the term identifier in the upper 8 bytes and the location in the lower 8 bytes, along with a fixed-width data field (e.g., one byte) that is carried along with the sort key. Additionally, the variable width data field case may be handled by a slower, more general radix sort that works with variable width fields.

**[0045]** FIG. 4A illustrates logic implemented for an index merge phase in accordance with certain implementations of the invention. FIG. 4B illustrates threading for merging sorted runs in accordance with certain implementations of the invention. In FIG. 4A, control begins at block 400 with the indexing component 142 scanning sorted runs from temporary storage. In block 402, the indexing component 142 performs a multi-way merge of the sorted runs. In the second phase (i.e., the index merge phase), the indexing component 142 uses, for example, a heap on top of each sorted run and performs a multi-way merge of the compressed sorted runs.

**[0046]** In block 404, the indexing component 142 performs remapping. It is possible that documents fed into the indexing component 142 have gaps in their rank value (e.g., documents 1, 6, 7, etc. are received, leaving a gap between ranks 1 and 6). The indexing component 142 remaps the rank locations to close up the gaps, ensuring that they are in sequence. In certain implementations, this is done by keeping a bit vector of the document identifiers that were put in the indexing component 142. In certain implementations, the bit vector is defined to be of the size of MaximumDocumentRank=50,000,000 bits so that the bit vector takes up about 48MB. At the start of the multi-way merge phase, a remapping table is built that maps each

document identifier to a new value. MaximumDocumentRank may be described as a value for a maximum document rank. In certain implementation, this table is MaximumDocumentRank*(4bytes) = 191MB in size. During the multi-way merge phase, each token document identifier is remapped to remove the gaps. In certain implementations, the remap table is not allocated until after the sort buffer (e.g., the second or third in-memory sort buffer referenced in blocks 308 and 318, respectively) has been deallocated.

**[0047]** In block 406, the indexing component 142 associates anchor text with corresponding content. That is, the indexing component 142 also brings together multiple pieces of a document if they are read from the data store at different times. In certain implementations, two document pieces are supported: a content section and an anchor text section. In certain implementations, the content section arrives and then the anchor text section arrives at a later time. This feature of the indexing component 142 is used to append anchor text to a document by giving anchor text locations in which the upper bit of the offset is set. Prior to indexing, the anchor text is extracted and a virtual document is built with the anchor text. This allows for sending the content of a document separately from the anchor text that points to the document. The content section and anchor text section may be correlated using a common document identifier.

**[0048]** In block 408, the indexing component 142 generates a final index with a dictionary, a block descriptor, and postings. FIG. 4C illustrates output of the indexing component 142 in accordance with certain implementations of the invention. In certain implementations, the indexing component 142 writes three types of output files: a dictionary file 460, a block descriptor file 462, and a postings file 464. The dictionary file 464 contains the terms, certain statistics for each term, and a pointer to the blocks that contain the term. The block descriptor file 462 contains pointers into blocks of the postings file 464 and certain additional information that is useful for random-access. The blocks in the postings file 464 may be the smallest unit of I/O performed on the postings file 464 (e.g., one block may be 4096 bytes). The postings file 464 contains raw posting list information for the terms and includes the location and data field, along with a

fast-access lookup table. The intra-block lookup table may be used to increase query performance. Without the lookup table, if an average random posting is in the middle of a block, then, accessing that posting would require starting from the first element in the block and decompressing and scanning each element prior to the desired posting. The lookup table allows accessing the location nearer to the block, which saves a large amount of unnecessary decompression work.

**[0049]** In FIG. 4C, the numbers in parenthesis are the number of bytes used to encode a particular field. For example, for fields labeled (VInt32), a compressed variable width byte encoding of a 32-bit number is used, and for fields labeled (VInt64), a compressed variable width byte encoding of a 65-bit number is used. In certain implementations, because the final files may grow large, there may be file splitting of final files.

**[0050]** Reordering posting lists in document rank order is useful so that documents with higher rank occur earlier in the posting list to allow for early termination of a search for terms using the index.

**[0051]** In FIG. 4B, during the merge phase, prefetching threads 432, 434 read data from the sorted runs in temporary storage 430 into a queue 436. A heap merge thread 438 is a multi-way merge thread that implements a heap merge, with results being stored in queue 440. A final index thread 442 outputs compressed index files to storage 444.

**[0052]** In certain implementations, 2-way Symmetric Multiprocessors (SMPs) may be utilized for the parallel processing of FIGs. 3D and 4B.

**[0053]** In certain alternative implementations, rather than using a sort-merge approach, the indexing component 142 uses a Move To Front (MTF) hash table to accumulate posting lists for terms in-memory. The MTF hash table allows a sort to be performed on the unique tokens (which are the hash keys), rather than on all occurrences. When an MTF hash table is used, dynamically resizable vectors are maintained to hold the occurrences of each term, which requires CPU and memory resources.

**[0054]** FIG. 5A illustrates a table 500 of benchmark test results for index build threading and hyperthreading in accordance with certain implementations of the invention. The benchmark test was run against an IBM x345 2-way SMP with dual 2.4 Ghz (gigahertz)

processors, with 4 Small Computer System Interface (SCSI) disks using software RAID 0. In the benchmark test, 500,000 documents were indexed, which is a 7BG document data store. The multithreaded indexer design efficiently used dual processors. In table 500, HT refers to HyperThread. For example, a 1 CPU with a 1 HyperThread CPU system increased performance by 18%, while a 2 CPU system increased performance by 44%.

**[0055]** FIG. 5B illustrates a table 510 of index build scaling performances in accordance with certain implementations of the invention. The data in table 510 indicates that indexing performance scales nearly linearly with index size. In table 510, the raw data store size was about 2.23 times larger than the index size.

**[0056]** Thus, the implementations of the invention provide a high performance, general purpose indexer designed for doing fast and high quality information retrieval. Certain implementations of the invention make use of a dual-path approach to indexing, using a high performance, pipelined radix sort. Additionally, implementations of the invention enable document rank ordering of posting lists, per-occurrence attribute information, positional token information, and the bringing together of documents in pieces (e.g., content and anchor text).

**[0057]** Thus, the sort for index creation takes as input a set of token occurrences (called postings) that include a token, the document the token occurs in, the offset within the document, and some associated attribute data. Postings are sorted by token, document, and offset, in that order, so the token is the primary sort key, the document is the secondary sort key, and the offset is the tertiary sort key. The data field is carried along through the sort process.

**[0058]** Implementations of the invention provide a high performance sort based on the fact that for many postings, the data field is of constant size. Using a fixed width token-ID to represent the token, the postings may be represented by a fixed width binary data structure. Also, the data field is a fixed width in many cases by construction. Therefore, a fast, fixed width sort may be used to sort the fixed width postings.

**[0059]** The postings that can not be handled by a fixed width sort are ones that have a variable width data field. These postings are sorted by a variable width sort. The results of both sorts are combined during a multi-way merge phase. In alternative implementations, the merge may be avoided, and the two sets of sorted postings may be maintained separately.

**[0060]** The sort keys are encoded so that, during the sorting process, the index is simultaneously ordered by term, terms are ordered by document ID, and offsets within documents are also ordered.

**[0061]** The offsets are encoded within a document so that multiple parts of a document are brought together automatically by the sort. Thus, the parts of a document may be fed into the index build separately. Conventional indexers require that documents be fed in as a whole.

**[0062]** In certain implementations, a radix sort may be used because the radix sort sorts in linear time for fixed width sort keys and is stable (i.e., preserves the original ordering of sort element when there are ties in a sort key). The stable sort property of radix sort is used to further improve indexing performance. In particular, if tokens in each part of a document are already in location order, sort is performed on the portion of the sort key that is needed to generate correct sort results.

**[0063]** If all tokens within a document are fed in order, the sort key need not use the offset portion, but can sort on token type, token, document identifier, and document section. If the document sections arrive in order, then the sort can be on token type, token, and document identifier. If the documents are in document rank order, the sort can be on token type and token.

**[0064]** Thus, the indexing component 142 may have different implementations, based on the order in which tokens, document sections, and documents arrive. For example, in certain implementations, the indexing component 142 recognizes that tokens of a given document are fed in order of appearance. In certain implementations, the indexing component 142 recognizes that for document sections, the content sections arrive before the anchortext sections. Also, in certain implementations, the indexing component 142

recognizes that the documents are fed in document rank order. Furthermore, in additional implementations, the indexing component 142 recognizes the order for two or more of tokens, document sections, and documents.

Additional Implementation Details

**[0065]** The described techniques for an architecture for an indexer may be implemented as a method, apparatus or article of manufacture using standard programming and/or engineering techniques to produce software, firmware, hardware, or any combination thereof. The term "article of manufacture" as used herein refers to code or logic implemented in hardware logic (e.g., an integrated circuit chip, Programmable Gate Array (PGA), Application Specific Integrated Circuit (ASIC), etc.) or a computer readable medium, such as magnetic storage medium (e.g., hard disk drives, floppy disks,, tape, etc.), optical storage (CD-ROMs, optical disks, etc.), volatile and non-volatile memory devices (e.g., EEPROMs, ROMs, PROMs, RAMs, DRAMs, SRAMs, firmware, programmable logic, etc.). Code in the computer readable medium is accessed and executed by a processor. The code in which various implementations are implemented may further be accessible through a transmission media or from a file server over a network. In such cases, the article of manufacture in which the code is implemented may comprise a transmission media, such as a network transmission line, wireless transmission media, signals propagating through space, radio waves, infrared signals, etc. Thus, the "article of manufacture" may comprise the medium in which the code is embodied. Additionally, the "article of manufacture" may comprise a combination of hardware and software components in which the code is embodied, processed, and executed. Of course, those skilled in the art will recognize that many modifications may be made to this configuration without departing from the scope of the present invention, and that the article of manufacture may comprise any information bearing medium known in the art.

**[0066]** The logic of FIGs. 2, 3A, 3B, 3C, and 4A describes specific operations occurring in a particular order. In alternative implementations, certain of the logic operations may

be performed in a different order, modified or removed. Moreover, operations may be added to the above described logic and still conform to the described implementations. Further, operations described herein may occur sequentially or certain operations may be processed in parallel, or operations described as performed by a single process may be performed by distributed processes.

**[0067]** The illustrated logic of FIGs. 2, 3A, 3B, 3C, and 4A may be implemented in software, hardware, programmable and non-programmable gate array logic or in some combination of hardware, software, or gate array logic.

**[0068]** FIG. 6 illustrates an architecture of a computer system that may be used in accordance with certain implementations of the invention. For example, client computer 100, server computer 120, and/or operator console 180 may implement computer architecture 600. The computer architecture 600 may implement a processor 602 (e.g., a microprocessor), a memory 604 (e.g., a volatile memory device), and storage 610 (e.g., a non-volatile storage area, such as magnetic disk drives, optical disk drives, a tape drive, etc.). An operating system 605 may execute in memory 604. The storage 610 may comprise an internal storage device or an attached or network accessible storage. Computer programs 606 in storage 610 may be loaded into the memory 604 and executed by the processor 602 in a manner known in the art. The architecture further includes a network card 608 to enable communication with a network. An input device 612 is used to provide user input to the processor 602, and may include a keyboard, mouse, pen-stylus, microphone, touch sensitive display screen, or any other activation or input mechanism known in the art. An output device 614 is capable of rendering information from the processor 602, or other component, such as a display monitor, printer, storage, etc. The computer architecture 600 of the computer systems may include fewer components than illustrated, additional components not illustrated herein, or some combination of the components illustrated and additional components.

**[0069]** The computer architecture 600 may comprise any computing device known in the art, such as a mainframe, server, personal computer, workstation, laptop, handheld

computer, telephony device, network appliance, virtualization device, storage controller, etc. Any processor 602 and operating system 605 known in the art may be used.

**[0070]** The foregoing description of implementations of the invention has been presented for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. It is intended that the scope of the invention be limited not by this detailed description, but rather by the claims appended hereto. The above specification, examples and data provide a complete description of the manufacture and use of the composition of the invention. Since many implementations of the invention can be made without departing from the spirit and scope of the invention, the invention resides in the claims hereinafter appended.